1 INTRODUCTION

Jealous stepmother and sisters; magical aid by a beast; a marriage won by gifts magically provided; a bird revealing a secret; a recognition by aid of a ring; or show; or what not; a dénouement of punishment; a happy marriage - all those things, which in sequence, make up Cinderella, may and do occur in an incalculable number of other combinations.

— MR. Cox ***1893***, *Cinderella: Three hundred and forty-five variants [**?**]*

The first web browser, Nexus, made its appearance in 1990 [**?**]. At its inception, web browsing consisted solely of retrieving and displaying small, static text pages. In other words, users could only read page content without any interactive components. However, the escalating computing power of devices, the proliferation of the internet, the valuation of internet-based companies and the demand for more engaging user experiences gave rise the concept of executing code in conjunction with web pages. In 1995, the Netscape browser revolutionized this concept by introducing JavaScript [**?**], a programming language that allowed code execution on the client-side. Interactive web content immediately highlighted benefits: unlike classical native software, web applications do not require installation, are always up-to-date, and are accessible from any device with a web browser. Significantly, since the advent of Netscape, all browsers offer JavaScript support. In the present day, the majority of web pages incorporate not only HTML but also JavaScript code, which is executed on client computers. Over the past several decades, web browsers have transformed into JavaScript virtual machines. They have evolved into intricate systems capable of running comprehensive applications, such as video and audio players, animation creators, and PDF document renderers.

JavaScript is presently the most widely utilized scripting language in all contemporary web browsers [**?**]. However, it is not without limitations due to the inherent characteristics of the language. For instance, each JavaScript engine needs the parsing and recompiling of the JavaScript code, resulting in substantial overhead. Just parsing and compiling JavaScript code consumes the majority of the load times of websites [**?**]. In addition to performance limitations,

[0] Compilation probe time 2023/11/07 18:30:15

JavaScript also has security concerns [**?**]. A notable example of this is the lack of memory isolation in JavaScript, which allows extraction of information from other processes [**?**]. These issues led the Web Consortium (W3C) to standardize a bytecode for the web environment in 2017, which is the WebAssembly (Wasm) bytecode. Hence, WebAssembly became the fourth official language for the web.

WebAssembly is designed with a focus on speed, portability, self-containment, and security [**?**]. It enables the ahead-of-time compilation of all programs from source languages such as C/C++ and Rust. Third-party compilers produce WebAssembly binaries, as is the case of LLVM. WebAssembly bytecode format abstracts its Instruction Set Architecture, making it akin to machine code instructions but independent of CPU architectures [**?**]. Resembling machine code, WebAssembly is already optimized and consists of consecutive binary sections. The contiguous array organization of a WebAssembly binary enables efficient processing, allowing compilers to speed up compilation through parallel parsing. Wasm binaries not only validate and compile rapidly but are also quick to transmit over a network due to their small sizes.

WebAssembly's versatility extends beyond web browsers to backend scenarios. Studies have highlighted the benefits of using WebAssembly as an intermediate layer, including improved startup times and enhanced memory usage [**?** **?**]. The Bytecode Alliance consequently proposed the WebAssembly System Interface (WASI) in 2019 [**?** **?**]. WASI standardized the execution of WebAssembly utilizing a POSIX system interface protocol, thus enabling WebAssembly's direct execution in the operating system.

1.1 WebAssembly security

WebAssembly is praised for its security, especially for its design that prevents programs from accessing data beyond their own memory. However, there has been less focus on potential vulnerabilities and attacks within WebAssembly's own memory [**?**].

Remarkably, WebAssembly binaries can have inherent vulnerabilities due to source code flaws. For example, the absence of stack-smashing protections like stack canaries in code compiled to WebAssembly could lead to undetected overflows in WebAssembly, causing crashes in standalone deployments [**?**] .

Moreover, significant risks exist from side-channel attacks on WebAssembly. Rokicki et al. revealed the risk for port contention side-channel attacks on WebAssembly binaries in browsers [**?**]. In standalone deployments, Genkin et al. demonstrated the potential for data extraction via cache timing-side channels in WebAssembly [**?**]. Similarly, Maisuradze and Rossow showed speculative execution attacks on WebAssembly binaries [**?**].

1.2 Software Monoculture

Web browsers and JavaScript have evolved significantly in the past thirty years, leading to numerous implementations. Yet, only Firefox, Chrome, Safari, and Edge are commonly used on devices. This situation reflects a software monoculture problem wherein a single flaw could impact multiple applications. The concept of monoculture is borrowed from biology and symbolizes an ecosystem at risk of extinction due to shared vulnerabilities and lack of diversity. Currently, web pages including WebAssembly binaries are centrally served from main datacenters. Thus, this monoculture issue is also applicable to the WebAssembly code served to web browsers. Therefore, sharing Wasm code through web browsers could also share its vulnerabilities.

The software monoculture problem exacerbates when considering the edge-cloud computing platforms and their adoption of WebAssembly to provide services. Specifically, in addition to browser clients, thousands of edge devices running WebAssembly as backend services could be affected by shared vulnerabilities. This scenario suggests that if one node in an edge network is vulnerable, all the others would be vulnerable in the exact same way since the same binary is replicated on each node. In other words, the same attacker payload could compromise all edge nodes simultaneously, meaning that a single distributed Wasm binary could trigger a worldwide attack.

1.3 WebAssembly malware evasion

WebAssembly is often used in browsers for computation-intensive activities, including gaming and image processing, but it has also been exploited by malicious actors for cryptojacking [**?**]. The popularity of WebAssembly for cryptojacking stems from its ability to execute a high volume of hash functions. Since WebAssembly outperforms JavaScript in speed, it is the natural option for cryptojacking. Besides, WebAssembly code's poor readability makes it a convenient tool for obfuscating harmful code. Cryptojacking via WebAssembly often involves a malicious JavaScript+WebAssembly payload that secretly executes on the victim's browser and generates passive income [**?**]. Because it is hard to detect and remove, cryptojacking can execute on a victim's computer, continuously using resources and generating income for the attacker.

Several techniques employ static analysis, dynamic analysis and even state-of-the-art machine learning methods to detect WebAssembly crytomalware [**? ? ? ? ? ?**]. Obfuscation studies have revealed weaknesses in several of these methods, indicating a largely unexplored threat to malware detection accuracy in WebAssembly. Yet, the majority of these studies do not consider the existence of obfuscation tools.

1.4 Problems statements

According to the discussion above, we identify three key problems to be addressed.

Ps1 WebAssembly security: WebAssembly ecosystem and binaries are vulnerable to attacks, specially side-channel threats. Existing WebAssembly research mostly reacts to existing vulnerabilities, leaving the potential for unidentified attacks. Besides, current defenses are limited to specific attacks or require the alteration of runtimes.

Ps2 Software monoculture: Identical WebAssembly binaries are deployed on multiple nodes and browsers. Deployment systems, including web browsers, might be also identical. Such a situation presents a potential threat to the entire ecosystem due to shared vulnerabilities.

Ps3 WebAssembly malware evasion: WebAssembly malware is a serious threat. Current implementated defenses are not sufficient to protect against WebAssembly malware, mostly because current defenses ignore malware obfuscation.

1.5 Software Diversification

This dissertation introduces tools, strategies, and methodologies designed to address the previously enunciated problem statements via Software Diversification. Software Diversification is a security-focused process that involves identifying, developing, and deploying program variants of a given original program [**?**]. Pioneers in this field, Cohen et al. [**?**] and Forrest et al. [**?**], proposed enhancing software diversity through code transformations. Their proposal suggested creating program variants while maintaining their functionalities to mitigate potential vulnerabilities.

Software diversification, as demonstrated in previous studies, can effectively remove vulnerabilities. For instance, Eichin et al. [**?**] presented seminal work in 1989, illustrating the practical benefits of diversification. Specifically, the diversification limited the Morris Worm's exploitation only to a few machines. However, despite extensive research, the use of software diversification in WebAssembly remains largely unexplored.

Software diversification could bolster WebAssembly analysis tools by incorporating diversified program variants, thereby hardening the attackers' task of exploiting vulnerabilities. Generated variants, created proactively for security, could emulate a broad spectrum of real-world conditions, subsequently improving the accuracy of WebAssembly analysis tools, including WebAssembly malware detectors. Moreover, current solutions to mitigate side-channel attacks on WebAssembly binaries either target specific attacks or need the modification of runtimes. Thus, by generating diversified variants independent of the

Contribution	Research papers			
	P1	P2	P3	P4
C1 Experimental contribution	✓	✓	✓	✓
C2 Theoretical contribution	✓		✓	
C3 Diversity generation	✓	✓	✓	✓
C4 Defensive diversification	✓	✓	✓	
C5 Offensive diversification				✓

Table 1.1: *Mapping between contributions and research papers .*

platform, software diversification could help address potential vulnerabilities in WebAssembly binaries. In the context of software diversification, we present the following, non-necesarilly orthogonal, contributions.

C1 Experimental contribution: For each proposed technique we provide an artifact implementation and conduct experiments to assess its capabilities. The artifacts are publicly available. The protocols and results of assessing the artifacts provide guidance for future research.

C2 Theoretical contribution: We propose a theoretical foundation in order to generate and improve Software Diversification for WebAssembly. We provide a formal definition of WebAssembly program variants and their diversity. We also provide a formal definition of WebAssembly program diversity generation.

C3 Diversity generation: We generate WebAssembly program variants. The variants are functionally equivalent to the original program, yet behaviorally diverse.

C4 Defensive Diversification: We assess how generated WebAssembly program variants could be used for defensive purposes. We provide empirical insights about the practical usage of the generated variants in preventing attacks.

C5 Offensive Diversification: We evaluate the potential for using generated WebAssembly program variants for offensive purposes. Our research includes experiments where we test the resilience of WebAssembly analysis tools against these generated variants. Furthermore, we offer insights into which types of program variants practitioners should prioritize to improve WebAssembly analysis tools.

1.6 Summary of research papers

This compilation thesis comprises the following research papers. In Table 1.1 we map the contributions to our research papers.

P1: **CROW: Code randomization for WebAssembly bytecode.**
Javier Cabrera-Arteaga, Orestis Floros, Oscar Vera-Pérez, Benoit Baudry, Martin Monperrus
Measurements, Attacks, and Defenses for the Web (MADWeb 2021), 12 pages
`https://doi.org/10.14722/madweb.2021.23004`

Summary: In this paper, we introduce the first entirely automated workflow for diversifying WebAssembly binaries. We present CROW, an open-source tool that implements software diversification through enumerative synthesis. We assess the capabilities of CROW and examine its application on real-world, security-sensitive programs. In general, CROW can create many statically diverse variants. Furthermore, we illustrate that the generated variants exhibit different behaviors at runtime.

P2: **Multivariant execution at the Edge.**
Javier Cabrera-Arteaga, Pierre Laperdrix, Martin Monperrus, Benoit Baudry
Moving Target Defense (MTD 2022), 12 pages
`https://dl.acm.org/doi/abs/10.1145/3560828.3564007`

Summary: In this paper, we synthesize functionally equivalent variants of deployed edge services. Service variants are encapsulated into a single multivariant WebAssembly binary. A random variant is selected and executed each time a function is invoked. Execution of multivariant binaries occurs on the global edge platform provided by Fastly, as part of a research collaboration. We demonstrate that multivariant binaries present a diverse range of execution traces throughout the entire edge platform, distributed worldwide, effectively creating a moving target defense.

P3: **Wasm-mutate: Fast and efficient software diversification for WebAssembly.**
Javier Cabrera-Arteaga, Nicholas Fitzgerald, Martin Monperrus, Benoit Baudry
Under review, 17 pages
`https://arxiv.org/pdf/2309.07638.pdf`

Summary: This paper introduces WASM-MUTATE, a compiler-agnostic WebAssembly diversification engine. The engine is designed to swiftly

generate functionally equivalent yet behaviorally diverse WebAssembly variants by randomly traversing e-graphs. We show that WASM-MUTATE can generate tens of thousands of unique WebAssembly variants in minutes. Importantly, WASM-MUTATE can safeguard WebAssembly binaries from timing side-channel attacks, such as Spectre.

P4: **WebAssembly Diversification for Malware evasion.**
Javier Cabrera-Arteaga, Tim Toady, Martin Monperrus, Benoit Baudry
Computers & Security, Volume 131, 2023, 17 pages

Summary: WebAssembly, while enhancing rich applications in browsers, also proves efficient in developing cryptojacking malware. Protective measures against cryptomalware have not factored in the potential use of evasion techniques by attackers. This paper delves into the potential of automatic binary diversification in aiming WebAssembly cryptojacking detectors' evasion. We provide proof that our diversification tools can generate variants of WebAssembly cryptojacking that successfully evade VirusTotal and MINOS. We further demonstrate that these generated variants introduce minimal performance overhead, thus verifying binary diversification as an effective evasion technique.

Thesis layout

This dissertation comprises two parts as a compilation thesis. Part one summarises the research papers included within, which is partially rooted in the author's licentiate thesis [**?**]. Chapter 2 offers a background on WebAssembly and the latest advancements in Software Diversification. Chapter 3 delves into our technical contributions. Chapter 4 exhibits two use cases applying our technical contributions. Chapter 5 concludes the thesis and outlines future research directions. The second part of this thesis incorporates all the papers discussed in part one.